LARGE SCALE OPTIMIZATION

David F. Gleich

April 21, 2026

These lecture notes are based on Nocedal and Wright, Chapter 7 as well as Griva, Sofer and Nash, Chapter 13

Consider the unconstrained optimization problem:

$$\text{minimize} \quad f(\mathbf{x})$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is twice continuously differentiable.

Throughout this entire section, we will assume that $f(\mathbf{x})$ and $\mathbf{g}(\mathbf{x})$ are both easy to evaluate. What does this mean, precisely? It means that your implementations of $f(\mathbf{x})$ and $\mathbf{g}(\mathbf{x})$ are both "fast" programs.[1] In a slightly more formal sense, we assume that the function and gradient correspond to $O(n)$ or $O(n \log n)$ computations.

1 WHY DO WE NEED LARGE-SCALE METHODS?

Let's consider the best methods we've studied for solving optimization problems: Netwon's method and the Quasi-Newton method. As a very small pseudo-code, and without appropriate checks on the non-singularity of $\boldsymbol{H}$, Newton's method is:

```
x_0 is given
while not done
  Solve H_k p_k = -g_k.
  Set x_{k+1} = x_k + α_k p_k using a strong Wolfe line search
```

First, doing the line-search with $\mathbf{x}_k$ as a 1,000,000 or even 5,000,000 dimensional vector is not a problem. All we need to do for a strong-Wolfe line search is compute inner-products with the search direction and the gradient. We have assumed the function and gradient are easy to evaluate so this isn't a problem.

However, this method involves solving a symmetric, positive definite linear system. We can do this via the Cholesky factorization in approximately $n^3/6$ floating point operation. If n is 100,000 or 1,000,000 – which are reasonable sizes for problems – then this computation is impractical using a textbook Cholesky method. We'll discuss the case of sparse Hessians, which is a case when it is reasonable to solve this problem, below.

If f is even modestly complicated, getting Hessian information may be difficult. Thus, let's also consider a Quasi-Newton BFGS method.

```
x_0 is given
T_0 is a scaled identity such that T_0 ≈ H(x_0).
while not done
  Set x_{k+1} = x_k - α_k T_k g_k using a strong Wolfe line search
  Set T_{k+1} = V_k T_k V_k + ρ s_k s_k^T
    where s_k = x_{k+1} - x_k, y_k = g_{k+1} - g_k
          ρ_k = (y_k^T s_k)^{-1},
          V_k = (I - ρ_k y_k s_k^T).
```

In this case, the first update to the Hessian inverse $\boldsymbol{T}_k$ results in a dense $n \times n$ matrix. If n is even 500,000, this problem is almost impossible to even store![2]

In a nutshell, the problem with both of these methods is that the linear algebra required does not scale. If your optimization problem is huge then you have three choices to solve it:

1. Use a simple method
2. Use scalable linear algebra
3. Change the method

We'll discuss these in turn.

[1] For instance, suppose that $\boldsymbol{A}$ is a large, sparse matrix. Then we can compute the matrix-vector product $\boldsymbol{A}\mathbf{x}$ quite easily using sparse linear algebra. Thus, for instance, suppose we want to solve a large, non-negative least squares problem and are willing to tolerate a barrier approximation to the constraint:

$$\text{minimize} \quad \|\boldsymbol{A}\mathbf{x} - \mathbf{b}\|^2 - \tau \mathbf{e}^T \log \mathbf{x}$$

then computing f and $\mathbf{g}$ can be done as follows:

```
function nonnegls(x,A,b,tau)
  y = A*x - b # efficient
  f = norm(y)^2 - tau*sum(log(x))
  g = 2*A'*y - tau*1.0./x
  return f, g # need to double-check
```

[2] The problem is symmetric, so we'd have to store "500,000 choose 2" double-precision numbers, or 931 GB of data; in theory this could be feasible.

2 SIMPLE METHODS

Since the problem with the methods is that the linear algebra does not scale, let's use methods with simple linear algebra. We've seen one such methods: gradient descent. Another method is conjugate gradients. This computes a new search direction that is conjugate to the previous search directions.

For both of these methods, all we need to do is to compute the gradient at each step. We have assumed that computing the gradient is easy (see above).

```
while not done
  x_{k+1} = x_k − α_k g_k
```

And so the only work is doing the line-search which is scalable under the assumptions above.

For conjugate gradients, the pseudo-code of the method is:

```
while not done
  ...
  β_k = ‖g_k‖^2/(g_k − g_{k−1})^T p_{k−1}  % O(n) work
  p_k = −g_k + β_k p_{k−1}  % O(n) work
  x_{k+1} = x_k + α_k p_k  % line-search
```

This method only involves $O(n)$ work beyond computing the line-search and gradients. The proof of convergence for CG involves only that every n iterations, we do a step of gradient descent as we reset the process.

The problem with both of these methods is that they only converge linearly to a solution. Unless the problems have very special structure where this results in a small amount of work, these methods tend to be slow.

Modern ML/AI methods There has been a lot of work on other simple, scalable optimization procedures for problems in machine learning and AI. These methods tend to be based on stochastic gradient descent. Almost all of these methods have linear convergence (or worse!), yet they are widely used in practice. The rationale for this usage is that the optimization problem is solving a problem for a particular sample of data. There is nothing perfect about this sample of data, and hence, we don't need to find the exact solution. Ideally, we'd like something that is nearby a robust / reliable minimizer to a solution that's easy to compute. In fact, this *approximation* is a form of regularization that helps the methods avoid overfitting the given data, although this behavior is much more complicated than you might expect. See research on the *double descent behavior* in . Thus, there are many large-scale problems where linear or even sub-linear convergence is perfectly acceptable.

3 SCALABLE LINEAR ALGEBRA

The key to scalable linear algebra is to exploit structure in the matrix. In a general, $n \times n$, symmetric matrix, there is no structure to exploit except for symmetry. Thus, scalable methods for these problems end up using many computers to store all the in the matrix. The linear algebra routines are parallel codes that use hundreds of machines to do things like compute a Cholesky factorization.[3]

[3] I've used the codes in ScaLAPACK, arguably the standard library, in order compute all eigenvalues and vectors of a symmetric 250,000-by-250,000 matrix using around 6000 processors.

3.1 SPARSITY

The most common structure in large problems is sparsity. It's a tricky concept to define, but the idea with sparsity is that most of the matrix is actually a zero entry. If the sparsity isn't too bad, then there has been an enormous amount of work on how to solve linear systems by exploiting sparsity. The key ideas here are how to minimize the number of new zeros introduced during a matrix factorization. Here's a really simple example.

Let $\boldsymbol{A} = \boldsymbol{I} + \mathbf{v}\mathbf{e}_1^T + \mathbf{e}_1\mathbf{v}^T$. This matrix has the shape of an arrow:

$$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & 0 & 0 & 0 & 0 \\ \bullet & 0 & \bullet & 0 & 0 & 0 \\ \bullet & 0 & 0 & \bullet & 0 & 0 \\ \bullet & 0 & 0 & 0 & \bullet & 0 \\ \bullet & 0 & 0 & 0 & 0 & \bullet \end{bmatrix}$$

where the $\bullet$ denotes a non-zero entry. We often just omit the zeros:

$$\begin{bmatrix} \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \\ \bullet & \bullet & & & & \\ \bullet & & \bullet & & & \\ \bullet & & & \bullet & & \\ \bullet & & & & \bullet & \\ \bullet & & & & & \bullet \end{bmatrix}.$$

If we compute the Cholesky factorization of this matrix, then we'll get a Cholesky factor:

$$\boldsymbol{A} = \boldsymbol{L}\boldsymbol{L}^T \quad \text{where} \quad \boldsymbol{L} = \begin{bmatrix} \bullet & & & & & \\ \bullet & \bullet & & & & \\ \bullet & \bullet & \bullet & & & \\ \bullet & \bullet & \bullet & \bullet & & \\ \bullet & \bullet & \bullet & \bullet & \bullet & \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}.$$

That is, we'll have take a very sparse matrix and made it entirely dense. If, however, we had permuted the matrix $\boldsymbol{A}$ such that: $\boldsymbol{A} = \boldsymbol{I} + \mathbf{v}\mathbf{e}_n^T + \mathbf{e}_n\mathbf{v}^T$ then we would have found:

$$\boldsymbol{A} = \begin{bmatrix} \bullet & & & & & \bullet \\ & \bullet & & & & \bullet \\ & & \bullet & & & \bullet \\ & & & \bullet & & \bullet \\ & & & & \bullet & \bullet \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix} \quad \text{with Cholesky factor} \quad \boldsymbol{L} = \begin{bmatrix} \bullet & & & & & \\ & \bullet & & & & \\ & & \bullet & & & \\ & & & \bullet & & \\ & & & & \bullet & \\ \bullet & \bullet & \bullet & \bullet & \bullet & \bullet \end{bmatrix}$$

The goal with sparse linear algebra is to figure out how to find permutations $\boldsymbol{P}$ that keep the Cholesky factors as sparse as possible. This procedure is very tricky and involves a host of beautiful relationships with graph theory. Tim Davis has an incredible book about it called "Direct Methods for Sparse Linear Systems"

For this reason, there has been a lot of work in Alex Pothen's group here at Purdue on how to compute sparse Hessians using automatic differentiation.

3.2 ITERATIVE METHODS & INEXACT NEWTON

Alternatively, if the problem will somehow let you compute matrix-vector product with the Hessian efficiently, then you could use an iterative method.[4] The goal would be to solve

$$\boldsymbol{H}_k \mathbf{p}_k = -\mathbf{g}_k$$

using only these matrix-vector product. As discussed in the book, if $\tilde{\mathbf{p}}_k$ is a direction where

$$\|\boldsymbol{H}\tilde{\mathbf{p}}_k + \mathbf{g}_k\| \leq \eta \|\mathbf{g}_k\|$$

and $0 < \eta < 1$, then we are "okay" – see Nocedal and Wright, Theorem 7.1.

[4] There are many problems where this is the case, even though the Hessian itself is dense, such as constrained least-squares problems where the constraint is a Fourier-transform matrix, which is dense, but has a $O(n \log n)$ matrix-vector product through the FFT.

Directions of negative curvature Quick aside! On why CG can fail and how that ends up helping us!

4 NEW METHODS: LIMITED-MEMORY BFGS

The goal in designing a new method is to exploit the structure of the BFGS Quasi-Newton method and generate a limited memory approximation of the Hessian with similar properties. The key idea follows from the question: *How could we exploit the fact that we think our optimization method will converge fast, such as in 10 steps?* If your routine only takes 10 iterations, then it seems crazy to do all the work in updating a Hessian approximation with $O(n^2/2)$ matrix-elements. In total, the 10 steps would generate a rank-20 change to the matrix, which would only take $20n$ matrix-elements to store. So the idea with a limited memory BFGS method is that we'll just store the last few changes to the Hessian as a low-rank update.

4.1 L-BFGS

In the BFGS update, we compute:

$$\boldsymbol{T}_{k+1} = (\boldsymbol{I} - \rho_k \mathbf{s}\mathbf{y}^T)\boldsymbol{T}_k(\boldsymbol{I} - \rho_k \mathbf{y}\mathbf{s}^T) + \rho \mathbf{s}_k \mathbf{s}_k^T.$$

Let $\boldsymbol{V}_k = (\boldsymbol{I} - \rho_k \mathbf{y}_k \mathbf{s}_k^T)$. Then

$$\boldsymbol{T}_{k+1} = \boldsymbol{V}_k^T \boldsymbol{T}_k \boldsymbol{V}_k + \rho \mathbf{s}_k \mathbf{s}_k^T.$$

Now, consider what happens if we store vectors $\mathbf{s}_k$ and $\mathbf{y}_k$ for the last 10 (or so) steps.

Then we have, starting from $\boldsymbol{T}_0$:

$$\begin{aligned}
\boldsymbol{T}_k = {} & \boldsymbol{V}_{k-1}^T \boldsymbol{V}_{k-2}^T \cdots \boldsymbol{V}_{k-m}^T \boldsymbol{T}_0 \boldsymbol{V}_{k-m} \cdots \boldsymbol{V}_{k-2} \boldsymbol{V}_{k-1} + \\
& + \rho_{k-m} \boldsymbol{V}_{k-1}^T \cdots \boldsymbol{V}_{k-m+1}^T \mathbf{s}_{k-m} \mathbf{s}_{k-m}^T \boldsymbol{V}_{k-m+1} \cdots \boldsymbol{V}_{k-1} + \\
& + \rho_{k-m+1} \boldsymbol{V}_{k-1}^T \cdots \boldsymbol{V}_{k-m+2}^T \mathbf{s}_{k-m+} \mathbf{s}_{k-m+1}^T \boldsymbol{V}_{k-m+2} \cdots \boldsymbol{V}_{k-1} + \\
& + \vdots \\
& + \rho_{k-1} \mathbf{s}_{k-1} \mathbf{s}_{k-1}.
\end{aligned}$$

To compute $\boldsymbol{T}_k \mathbf{g}_k$ all we need to do is multiply $\boldsymbol{T}_k$ by a vector. We don't actually need the elements itself! We can do that using the following algorithm:

$\mathbf{q} = \mathbf{z}$
for i=$k-1$ to $k-m$
  $\alpha_i = \rho_i \mathbf{s}_i^T \mathbf{q}$ % need to save these **for** below
  $\mathbf{q} \leftarrow \mathbf{q} - \alpha_i \mathbf{y}_i$
$\mathbf{r} = \boldsymbol{T}_0 \mathbf{q}$
for i=$k-m$ to $k-1$
  $\beta = \rho_i \mathbf{y}_i^T \mathbf{r}$
  $\mathbf{r} \leftarrow \mathbf{r} + \mathbf{s}_i(\alpha_i - \beta)$

Then we'll have $\mathbf{r} = \boldsymbol{T}_k \mathbf{z}$.

So the idea with Limited Memory Quasi Newton (L-BFGS) is that we store the last T updates to the $\boldsymbol{T}_0$ and use those to estimate $\boldsymbol{T}_k \mathbf{g}_k$. We can actually change $\boldsymbol{T}_0$ at each step, and so a common choice is $\boldsymbol{T}_0 = \mathbf{s}_{k-1}^T \mathbf{y}_{k-1} / (\mathbf{y}_{k-1}^T \mathbf{y}_{k-1}) \boldsymbol{I}$. This choice attempts to estimate the norm of the Hessian along the most recent search direction.

In practice, you keep the most recent m iterates around. Usually these are stored in a circular buffer.